《编译原理》（第 2 版）勘误表

2008-8-31

1、第 2 页倒数第 2 行改成：

分隔单词的空格通常在词法分析时被删去。

2、第 3 页图 1.2 改成：

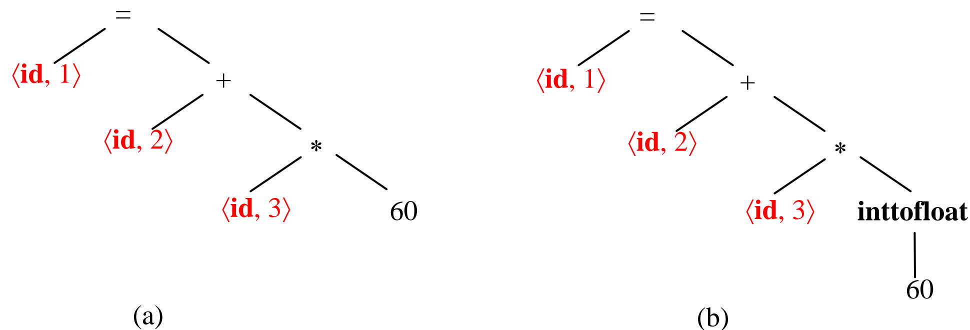


图 1.2 语义分析插入了类型转换

3、第 16 页倒数第 9 行开始的那段改成：

上一节提到，字符串集合由叫做模式的规则来描述。正规式是表示这些规则的一种重要方法，因此本节围绕正规式来介绍记号的描述与识别。在介绍正规式前，先给“语言”一个形式定义。

4、第 18 页表 2.3 的第 4 行第 4 列改成：

ε 肯定出现在一个闭包中

5、第 26 页图 2.10 上面的那段改成：

ε-*closure*(T)的计算是从给定的结点集合出发，在图上搜索可达结点的典型过程。该图只包含 NFA 的含ε 标记的边，T 是给定的结点集合。计算ε-*closure*(T)的简单算法是用栈来保存那些边还没有完成ε 转换检查的状态。图 2.11 描述了这样的过程。

6、第 43 页倒数第 5 行第一句改成：

正规式可以描述的语言都能用上下文无关文法来描述。

7、第 61 页 5 行开始的那段改成：

（2）仅使用 FOLLOW(A)作为 A 的同步集合是不够的。例如，分号在 C 语言中作为语句的结束符，那么作为语句开始符号的关键字没有出现在表达式非终结符的 FOLLOW 集合中。这样，仅按上面（1）来设定同步记号集合的话，作为赋值结束的分号的遗漏会引起下一语句的开始关键字被跳过。

8、第 74 页图 3.15 第 2 行改成：

$C = \{closure\ (\{[S' \to \cdot\ S\ ]\})\}$；

9、第 80 页图 3.19 倒数第 7 行改成：

置 C 的初值为$\{closure(\{[S' \to \cdot\ S, \$]\})\}$；

10、第 111 页倒数第 4 行开始的那段改成：

语法树作为一种中间表示，允许把翻译从分析中分离出来，形成先分析后翻译的方式，即先分析生成语法树，然后再基于语法树进行翻译。即使是边分析边翻译，语法树作为一种概念上的中间表示，也是有用的。C 和 Java 的编译器通常显式构造语法树。

11、第 119 页图 4.9 下第 1 行第 1 句改成：

图 4.10 给出了图 4.9 的动作是怎样为 $a*5*b$ 构造语法树的。

12、第 128 页倒数第 9 行开始的那段改成：

(a) 写一个翻译方案，它输出每个 a 的嵌套深度。例如，对于句子$(a, (a, a))$，输出的结果是 1　2　2。

13、第 134 页倒数第 11 行和倒数第 7 行，第 135 页第 7 行（两处）、第 9 行，第 136 页倒数第 3 行，第 137 页第 2 行，第 149 页倒数第 6 行：

“形式化的”都改成“形式的”

14、第 139 行倒数第 3 条规则改成：

$$\text{(Exp Index)} \qquad \frac{\Gamma \vdash E_1 : array(N,T), \quad \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1[E_2] : T} \quad (0 \le E_2.val \le N-1)$$

15、第 141 页倒数第 12 行改成：

$S \to \mathbf{id} := E$ 　　　　　$\{\mathbf{if}\ (\mathbf{id}.type == E.type\ \&\&\ E.type \in \{boolean, integer\})$

16、第 154 页图 5.13 改成：

cell = *record*
,
:　:
info　*integer*　next　*pointer*
cell
(a)

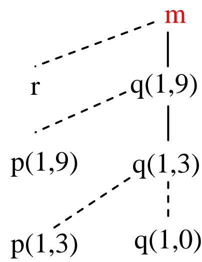
cell = *record*
,
:　:
info　*integer*　next　*pointer*
(b)

图 5.13　递归定义的类型名

17、第 157 页表 5.4 右栏的倒数第 4 行改成：

$S = \{s \mid s \in E_2.types \text{ and } s \to t \in E_1.types\}$　（备注：改成斜体）

18、第 170 页图 6.9 最后一行左栏改成：



19、第 172 页第 7 行开始的那段改成：

（4）q 根据局部数据域和临时数据域的大小来减小 *top_sp* 的值，也就是进行局部数据和临时数据的空间分配，并初始化它的局部数据，开始执行过程体，如图 6.10 所示。

20、第 203 页中间那段改成：

静态单赋值形式（*static single-assignment form*，简称 SSA）是一种便于某些代码优化的中间表示。SSA 有两个显著特点可区别它和三地址代码。第一个特点是，SSA 中所有赋值指令都是对不同名字变量的赋值，所有才有静态单赋值这个术语。图 7.4 是用三地址代码和静态单赋值形式写的同一个中间语言程序，注意，在 SSA 表示中，下标用来区别变量 p 和 q 的每个定义。

21、第 204 页图 7.5 最后 1 行改成：

$T \rightarrow \uparrow T_1$　　　　　$\{T.type = pointer\ (T_1.type);\ T.width = 4;\}$

22、第 206 页第 2 行改成：

$M \rightarrow \varepsilon$　　　　　$\{t = mkTable(nil);\ push(t,\ tblptr);\ push(0,\ offset);\}$

23、第 207 页图 7.8 中最后 1 行改成：

$L \rightarrow \varepsilon$　　　　　$\{t = mkTable(\text{nil});\ push(t,\ tblptr);\ push(0,\ offset);\}$

24、第 210 页倒数第 2 段前 2 行改成：

使用 *Elist* 的综合属性 *array* 来传递符号表中数组名条目的指针。并使用 *Elist.ndim* 来记录已分析过的下标表达式的个数。函数 *limit*(*array*, *j*)返回 n_j ，它是 *array* 指向数组第 *j* 维的大小。

25、第 211 页倒数第 6 行改成：

(7)　$Elist \rightarrow Elist_1,\ E$　　$\{t = newTemp();\ m = Elist_1.ndim + 1;$

26、第 215 页图 7.12 下面那段改成：

图 7.12 中的 *B.true*，*B.false*，*S.begin* 和 *S.next* 都是三地址指令的标号，它们都是继承属性。图(a)为 if-then 的结构图，由于 *B.code* 究竟有多少条指令需等 *B* 翻译结束才知道，因此在翻译 *B* 的过程中难以知道 *B.true* 在三地址指令序列中的准确位置，为便于翻译，采用给三地址指令加标号的方式，函数 *newLabel* 每次调用时返回一个新的标号。

27、第 230 页第 13 行改成：

的二地址指令，其中 *op* 是操作码，源和目的都是数据域。该机器有如下的操作码：

28、第 241 页表 8.4 最后 1 行改成：

*p = a	MOV a, *Rp	2	MOV Mp, R MOV a, *R	4	MOV a, *Sp(Rs)	3

29、第 259 页倒数第 2 段改成：

通常，明了所有路径上的所有程序状态是不可能的。数据流分析不区分到达一个程序点的不同路径，也不掌握完整的状态，而且它提炼出某些细节，以获取用于分析目的的数据。下面两个例子说明怎样从一个点的状态中提炼出不同的信息。

30、第 261 页倒数第 1 段改成：

如果存在从对 x 的定值 d 之后那个点到点 p 的一条路径，并且在这条路径上没有对 x 的定值，那么称定值 d 到达点 p。如果在这条路径上其他某个地方有对 x 的定值，那么称变量 x 在 d 的定值被**注销**。直观上说，如果某个变量 x 在 d 的定值到达点 p，并且运行时在点 p 引用 x，则 d 可能是 x 最近一次定值的位置。

31、第 270 页倒数第 2 行改成：

半格可能还有底元，它用⊥来指称，使得下式成立：

32、第 275 页倒数第 6 行改成：

（3）如果框架单调并且半格的高度有限，那么可以保证算法收敛。

33、第 277 页倒数第 11 行第 1 句改成：

注意，如果一个流图包含环的话，MOP 解所考虑的路径数是无界的。

34、第 277 页倒数第 1 行最后 1 句改成：

从某种意义上说，可以认为⊤代表没有任何信息。

35、第 280 页第 2 行开始的那段改成：

下面证明常量传播框架是单调的。首先考虑 f_s 在一个变量上的影响。除了 s 的右部是 $y + z$ 种情况以外，对于其他所有情况，f_s 不改变 $m(x)$的值，或者改变这个映射到返回一个常量。在这些情况下，f_s 肯定是单调的。

36、第 281 页第 1 行改成：

$$f_3(f_1(m_0) \wedge f_2(m_0)) \prec f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

37、第 286 页倒数第 16 行开始的第 1 句改成：

总的来说，表达式的预期用来约束表达式的最早放置，表达式的放置不能再提早到它不具备预期性质的位置。

38、第 286 页倒数第 2 段改成：

下面给出完整的惰性代码移动算法。为了使算法简洁，假定最初每个语句构成一个只含本身的基本块，并且只在块的入口放置表达式计算的副本。为了保证这种简化不会降低这种技术的效力，将新基本块添加在一条边的源结点和目的结点之间，如果该目的结点的前驱多

于一个。这样做明显关照到了程序中所有的关键边。

39、第 288 页图 9.25 改成：（注意左图中 B_3 和 B_5 改成灰色）

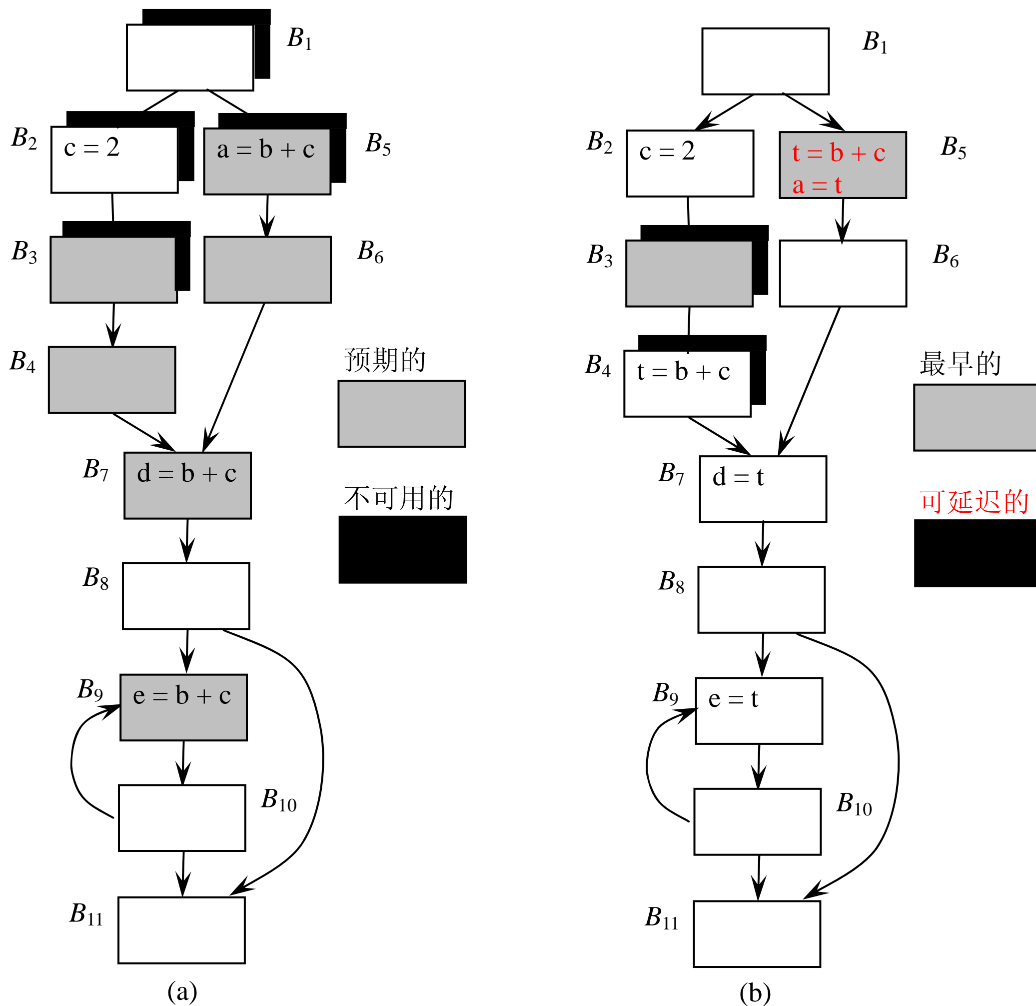


图 9.25 例 9.18 的流图

40、第 288 页倒数第 9 行开始的那段改成：

在第二步的结尾，表达式的副本将被放置在该表达式最早被期望的程序点。要做到这一点，需要定义可用表达式数据流问题。一个表达式在点 p **可用**，如果在到达点 p 的所有路径上它都被期望。这里定义的问题和 9.2.5 节描述的可用表达式类似，区别在于迁移函数。一个表达式在一个基本块的出口可用，如果该表达式在入口可用或者在入口的预期表达式集合中（即如果选择在这里计算它，则它就成为可用的了），并且没有被该基本块注销。

41、第 289 页第 4 行开始的那段改成：

例 9.20 在图 9.25(a)中，有黑阴影的块表示表达式 b + c 在入口不可用，它们是 B_1，B_2，B_3 和 B_5。最早放置由带黑阴影的灰色块表示，它们是 B_3 和 B_5。例如，b + c 在 B_4 的入口是可用的，因为存在路径 $B_1 \to B_2 \to B_3 \to B_4$，并且 b + c 在 B_3 是被预期的，还有从 B_3 的入口开始，b 和 c 都没有被重新计算。

42、第 306 页倒数第 6 行第 1 句改成：

本章最后一节简要介绍使用数组的计算密集型程序在多处理器系统上的优化问题。

43、第 307 页第 12 行第 1 句改成：

在考虑指令级并行时，通常想象成一个处理器在单个时钟周期内发射几个操作。

44、第 308 页倒数第 1 行第 1 句改成：

更复杂的代码调度器能够乱序执行指令。

45、第 309 页第 12 行开始的第 1 和 2 句改成：

这些调度约束保证优化程序和原程序产生同样的结果。但是，因为代码调度会改变操作执行的次序，因此在一个程序点的内存状态可能与顺序执行的任何内存状态都不匹配。

46、第 312 页最后 1 句改成：

10.5 节将讨论在特定调度算法中，寄存器分配和代码调度之间的相互影响。

47、第 314 页第 1 行改成：

许多高性能处理器提供专门的特性来支持投机地访问内存，下面介绍其中最重要特性。

48、第 314 页第 12 行开始的那段改成：

由于分支的花费很大，尤其是在分支预测出现错误时。判定指令（*predicated instruction*）可用来减少程序中的分支数。判定指令与普通指令类似，但是有一个额外的判定操作数来看守它的执行。只有在判定为真的情况下，判定指令才执行。

49、第 314 页倒数第 9 行开始的那段改成：

判定执行伴随着一些代价的发生。判定指令被预取和译码，即使它们有可能最后不被执行。静态调度必须预留所有需要用于它们执行的资源，并保证所有潜在的数据相关都满足。判定执行不应该被大量使用，除非机器有足够的资源。

50、第 318 页第 11 行第 1 句改成：

对于有适度指令级并行的机器，仅考虑紧凑单个基本块的调度会引起许多资源空闲。

51、第 321 页倒数第 12 行开始的那段改成：

例 10.9 在图 10.8 中的流图中，两个对 x 的赋值中的一个可以移动到最上面的那个基本块，因为该变换能维持原来程序中的所有相关性。但是，如果其中一个对 x 的赋值被上移，则另一个就不能移动了。具体来说，移动前变量 x 在最上面块的出口不是活跃的，移动后成为活跃的。如果一个变量在某个程序点活跃，则不能把对该变量的投机定值移到该程序点的上面。

52、第 322 页第 8 行开始的那段改成：

在一些全局调度算法中，循环迭代的边界是代码移动的一种屏障。一次迭代中的操作不可以和另一次迭代中的操作有任何重叠。打破这种屏障的一种简单而非常有效的技术是在代码调度前把循环展开几次迭代，使循环体中有更多的指令，从而全局调度算法有机会发现更多的并行性。图 10.9 给出了 for 循环展开的示例。

53、第 322 页 10.4.6 节的标题改为（目录中也应相应修改）：

静态调度器和动态调度器的相互影响

54、第 324 页第 6 行开始的那段改成：

例 10.11 在例 10.10 的例子中，虽然很难发现在一次迭代中的并行性，但是却有很多穿越迭代的并行性。循环展开把循环中几次迭代的代码放在一个大基本块中，然后使用表调度算法使这些操作并行执行。如果先把循环展开 4 次迭代，然后把算法 10.1 用到展开后的代码，得到的调度表见图 10.11（为简单起见，忽略了寄存器分配的细节）。展开后每次迭代的执行用 13 周期，即原来的每次迭代仅需要 3.25 周期。

55、第 326 页前 3 行改成：

用局部最优调度，启动间隔就不得不加长到 4 周期，以避免资源冲突，那么吞吐能力就降到一半。该例说明流水线调度的一个重要原则：必须仔细安排调度以最优化吞吐能力。局部紧凑的调度虽然极小化了单次迭代的时间，但把它流水化时可能得不到最理想的吞吐能力。

56、第 329 页倒数第 5 行的那句改成：

如何解决这两者之间的相互依赖？

57、第 331 页倒数第 8 行开始的那段改成：

本节首先介绍并行计算机系统结构的概况，然后给出并行化的基本概念、程序循环的变换和对并行化有用的概念，再讨论类似的考虑怎样用于优化数据局部性，最后以矩阵乘算法的优化为例。

58、第 332 页倒数第 8 行开始的那句改成：

类似于存取速度越快则容量越小的内存分层设计原则，支持处理器之间快速通信的机器只有少量处理器。

59、第 335 页例 10.18 改成（调整缩进）：

```
for (i = 0; i < n; i++) {
    Z[i] = X[i] − Y[i];
}
for (i = 0; i < n; i++) {
    Z[i] = Z[i] * Z[i];
}
```

60、第 335 页倒数第 3 行的第 1 句改成：

例 10.17 被融合循环的性能较好，因为它有较好的数据局部性。每个差计算出来后就立即计算它的平方。

61、第 337 页倒数第 10 行第 1 句改成：

但是，当使用 X 的完整一行时，该算法需要逐列访问 Y 的所有元素。

62、第 339 页倒数第 18 行改成：

每一块实际只需要取到缓存中一次，所以和 10.6.5 小节对基本算法的分析一样，在这里不考

虑由

63、第 339 页倒数第 14 行改成：
阵乘法需要 n^3 次乘加计算，所以取一对块到缓存的操作的总次数是 n^3/b^3。由于对于 X 和 Y 的一对块

64、第 339 页倒数第 11 行改成：
整的矩阵都能装到缓存，那么缓存未命中的数量是 $O(n^2/c)$。在这种情况下，可以取 $b = n$，即让整

2009-5-5

1、第 129 页第 10 行改成：
4.14 参照 4.3.3 节，给出例 4.10 中对应非终结符 T 的翻译函数。

2、第 204 页倒数第 9 行改成：

$D\ ;\ S$

3、第 205 页第 4 行改成：

$P \rightarrow D\ ;\ S$

4、第 206 页第 1 行改成：

$P \rightarrow M\ D\ ;\ S \qquad \{addWidth(top(tblptr), top(offset)); pop(tblptr); pop(offset);\}$

5、第 271 页第 9 行改成：

对 V 中所有的 x 和 y，$x \le y$ 当且仅当 $x \wedge y = x$。

6、第 274 页倒数第 13 行改成：

$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$

7、第 274 页倒数第 9 行改成：

$(y \cup z) - K = (y - K) \cup (z - K)$

8、第 278 页第 3 行改成：

$\mathrm{MOP}[B_4] = (f_{B_3} \circ f_{B_1})(v_{\mathrm{ENTRY}}) \wedge (f_{B_3} \circ f_{B_2})(v_{\mathrm{ENTRY}})$

9、第 299 页倒数第 4 行改成：

试问应该怎样修改 9.4 节的框架？

2009-9-25

1、第 25 页图 2.8 改成

$s = s_0;$
$c = nextChar(\,);$

```
while (c != eof) {
    if (move(s, c)未定义) return “no”; else s = move (s, c);
    c = nextChar();
}
if (s 属于 F) return "yes";
else return "no" ;
```

2009-10-15

1、第 56 页第 2 行改成

如果显式地维持一个栈，而不是隐式地通过递归调用，那么可以构造非递归的预测分析器。

2、第 64 页第 3 行改成

$$E \Rightarrow_{rm} \underline{E * E}$$

3、第 64 页第 13 行改成

$$\Rightarrow_{rm} \underline{E * E} + \mathbf{id}_3$$

2009-10-24

1、第 79 页第 2 行改成

$S \Rightarrow V = E \Rightarrow *E = E$），第二项目使得 $action[2, =]$为按 $E \rightarrow V$ 归约。…

2、第 80 页图 3.19 第 5 行改成

for(FIRST(βa)的每个终结符 b)

3、第 71 页倒数第 3 行改成

LL 方法和 LR 方法有明显的区别。…

4、第 102 页倒数第 6 行改成

3.17 为习题 3.3 的文法构造 SLR 分析表。

5、第 52 页第 6 行

本节首先介绍自上而下分析的基本概念和一般方法，然后定义适合于自上而下分析的 LL(1)

6、第 142 页图 5.3 的名称改成

图 5.3 从整型到实型的类型转换

7、第 160 页习题 5.18 第 1 行改成

5.18 对于下面的 C 语言程序，在 x86/Linux 机器上，编译器报告第 11 行有错误:

8、第 152 页图 5.12 的第 11 行改成

return sequiv(s_1, t_1) && sequiv(s_2, t_2);

9、第 172 页第 3 行改成

在这些操作过程中，*top_sp* 的值在不断减小。

10、第 182 页最后一段改成

无用单元收集是一个在堆上寻找空间的过程，它寻找程序不再使用因而可以重新分配给其他数据项的空间。对 Java 语言来说，存储块的释放靠无用单元收集器来完成的，因此无用单元收集器是 Java 运行系统中完成内存管理的重要子系统，它将在 11.3 节介绍。

11、第 200 页第 12 行改成

例如，(8 – 4) + 2 的后缀表示是 8 4 –2 +，而 8 – (4 + 2)的后缀表示 8 4 2 + –。

12、第 218 页的前 6 行改成

例 7.3 考虑语句

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y – z
```

2010-1-18

1、第 56 页算法 3.1 的前两行改成

输入　串 *w* 和文法 *G* 的分析表 *M*。

输出　如果 *w* 属于 *L*(*G*)，则输出 *w* 的最左推导，否则报告错误。

2、第 96 页倒数第 15 行改成

```
lines   : lines expr '\n'        {printf ( "%g \ n" , $2 );}
```

3、第 105 页习题 3.41 的程序改成（变成 8 行）

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        return q
    else
        return gcd(q, p%q);
}
```

4、第 169 页第 7 行改成

…因此它们的生存期不遵守栈式规则…

5、第 182 页倒数第 9 行改成

…生存期没有被约束在创建它们的过程活动的生存期之内…

6、第 201 页表 7.1 最后 1 行左栏改成

$E \to \textbf{id}$

7、第 209 页第 8 行改成

二维数组通常用两种形式之一存储：行为主（一行接一行）或列为主（一列接一列）。

8、第 221 页第 20 行改成

$emit$('call', **id**.$place$, n) }

9、第 237 页第 1 行改成

假如反向扫描到达三地址语句 $i : x = y\ op\ z$，执行下面几步：

2010-3-27

1、第 51 页倒数第 6 行改成

（3）$CB \to BC$ 用 $n\times(n-1)/2$ 次，交换相邻的 CB，得到 $S \Rightarrow^+ a^nB^nC^n$；

2、第 64 页第 6 行改成

$\Rightarrow_{rm} E * \underline{\textbf{id}_2} + \textbf{id}_3$

3、第 70 页两张表之间的第 6 行改成

动作是按 $F \to \textbf{id}$ 归约。这时两个符号（状态符号 5 和文法符号 **id**）弹出栈，状态 0 显露出来。

4、第 84 页第 14 行改成

并后变成$\{[A\to c\cdot, d/e], [B\to c\cdot, d/e]\}$，出现归约-归约冲突，因为面临 e 或 d 时，不知道应该

5、第 100 页倒数第 2 行改成

$S \to (\ L\ ) \mid a$

2010-5-16

1、第 268 页倒数第 8 行改成

用 I^j 和 O^j 分别表示 IN[B_2]和 OUT[B_2]的第 j 次近似，这些方程可以重写成递推的形式

2、第 268 页倒数第 4 行改成

以 $O^0 = \varnothing$开始，则得到 $I^1 = \text{OUT}[B_1] \cap O^0 = \varnothing$。但是，如果从 $O^0 = U$ 开始，则得到 $I^1 = \text{OUT}[B_1] \cap O^0 = \text{OUT}[B_1]$，这才是应该得到的。直观上说，以 $O^0 = U$ 开始，所得到的解更有希望，因为它正确反映了一个事实：OUT[B_1]中没有被块 B_2 注销的表达式在块 B_2 的出口可用。 □

2010-5-22

1、第 293 页第 8 行改成

一个图的深度优先搜索访问该图所有结点各一次，它从图的起点开始并且尽快地访问离起点尽可能远的结点。

2、第 293 页第 12 行开始的那一段改成

深度优先排序是一种对流图分析来说很重要的排序，它是后序遍历的逆。深度优先排序先访问一个结点，然后从它的最右子结点开始，逐步向左遍历各结点，直到最左子结点。在为流图构建相应的树之前，需要确定流图中每个结点的后继中，哪个被看成该树上它的最右子结点，哪个次之，直到哪个是最左子节点。

3、第 174 页第 2 行改成

而不是堆上的理由是避免垃圾收集器收集它们的开销，只有那些局部于一个过程并在该过程返回后不可访问的数据对象才能分配在栈上。

2010-10-9

1、第 14 页表 2.1 第 4 行改成

relation **<, >, <=, >=, =, <>** < 或 > 或 <= 或 >= 或 = 或 <>

2、第 9 页倒数第 4 行改成

因为现在计算机系统的性能不仅仅取决于它的原始速度（*raw speed*）

3、第 122 页第 15 行改成

$T \rightarrow$ **char** $\{T.type = char;\}$

2011-03-13

1、第 109 页第 2 段

图 4.2 给出句子 **int** $\textbf{id}_1$, $\textbf{id}_2$, $\textbf{id}_3$ 的注释分析树。这些属性值的计算次序是，首先计算根的左子结点的属性 *T.type*，然后在根的右子树中自上而下地计算三个 *L* 结点的 *L.in*。在每个 *L* 结点还调用过程 *addType*，在符号表中将右子结点上标识符的类型记为整型。

改成

图 4.2 给出句子 **int** $\textbf{id}_1$, $\textbf{id}_2$, $\textbf{id}_3$ 的注释分析树（仅注释了部分属性）。这些属性值的计算次序是，首先计算根的左子结点的属性 *T.type*，然后在根的右子树中自上而下地计算三个 *L* 结点的 *L.in*。在每个 *L* 结点还调用过程 *addType*，在符号表中将 **id** 子结点上标识符的类型记为整型。

2011-3-16

1、第 121 页图 4.12 的第 2 行改成

syntaxTreeNode *nptr, *i1, *s1, *s;

2、第 185 页倒数第 12 行

即使知道了哪些指令会被频繁执行，最快的缓存也可能无法把它们同时放在其中。

改成

即使知道了哪些指令会被频繁执行，最快的缓存也可能没有大到足以把它们同时放在其中。

3、第 200 页倒数第 12 行改成

后缀表示也可以拓广到表示赋值语句和控制语句，但很难用栈来描述控制语句的计算。

2012-4-6

1、第 147 页倒数第 11 行改成：

（2）对任何其他满足 $S'(t_1) = S'(t_2)$的代换 S'，代换 $S'(t_1)$是 $S(t_1)$的实例。